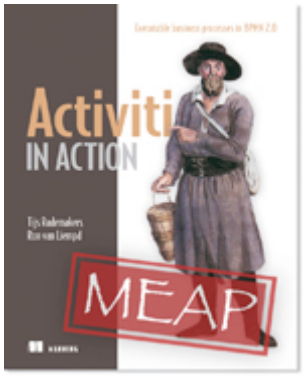
MANNING PUBLICATIONS

Developing the business logic

An article based on



Activiti in Action

EARLY ACCESS EDITION

Executable business processes in BPMN 2.0
Tijs Rademakers and Ron van Liempd
MEAP began: December 2010
Softbound print: Fall 2011 | 475 pages
ISBN: 9781617290121

This article is taken from the book Activiti in Action*. The authors explain business logic tasks involved in a loan request process.*

In this article, we will focus on the business logic tasks that need to be implemented in a loan request process. The part of the loan request process we will cover in this article can be seen in figure 1.

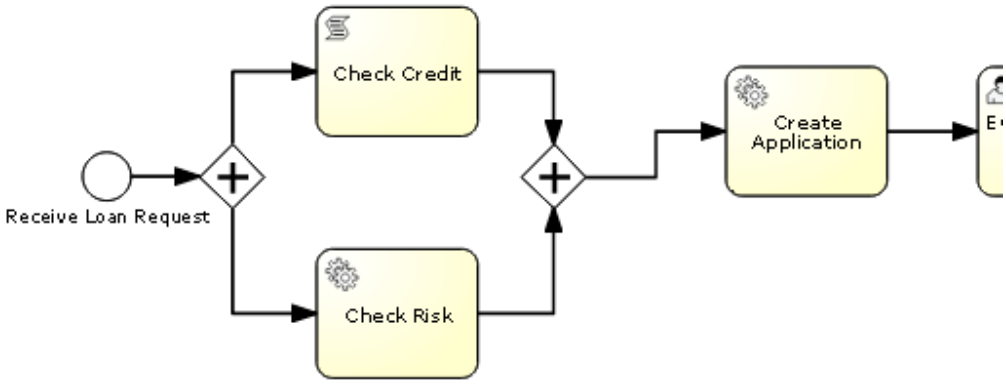


Figure 1 Fragment of Activiti Modeler containing the first part of the loan request process

First, we take a look at the script task and, then, we implement the Java service tasks. Finally, at the end of this section, we show the BPMN 2.0 XML we created so far and test the process with Eclipse.

Implementing a script task

The first task we encounter when we look at the loan request process is the script task. We will use it to implement the credit check.

Understanding script tasks in BPMN 2.0

The script task is an official BPMN 2.0 construct. In figure 1, you can see the symbol that BPMN 2.0 prescribes for the script task. It has the same rectangular shape as a regular service task. The little graphical marker in the upper left corner indicates that the task is a script task. The script that is defined in the task will be executed by the process engine—in our case, the Activiti Engine. An analyst will define the task in the model and a developer has to implement the script with a language the engine can interpret. When execution of the script is completed, the script task itself will also complete, and the engine moves toward the next execution.

Important BPMN 2.0 attributes of the script task construct are `scriptFormat` and `script`. The `scriptFormat` attribute defines the format of the script and is mandatory. The optional `script` attribute contains the actual script that needs to be executed. If no script is defined, the task will just complete without doing anything.

Working with Script tasks in Activiti

For the Activiti Engine to execute the coded script, the `scriptFormat` attribute must have a value that is compatible with the JSR-223, *Scripting for the Java platform*. The scripting languages that come with engines that conform to the JSR are numerous. We will name a just a few of the languages that are supported: Groovy, Jaskell, AWK, Python, JavaScript, and PHP. For more information, you can check out the JSR-223 specification at http://jcp.org/en/jsr/detail?id=223.

Because, by default, the Groovy jar is shipped with the Activiti distribution, we will use Groovy as the script language in the Check Credit task. If you want to use another JSR-223-compatible scripting engine, it is sufficient to add the corresponding jar file to the classpath and use the appropriate name in the script task configuration.

All the process variables are standard accessible in the script, because the script has access to the execution that arrives in the task. You can, for example, use the process variable `inputArray`, an array of integers, as shown below.

```
<script>
    Sum = 0;
    for ( i in inputArray ) {
      sum += i
    }
</script>
```

That's great stuff isn't it? Besides reading variables, it is also possible to set process variables in a script by using an assignment statement. In the example above, the `sum` variable will be stored as a process variable after the script has been executed. If you want to avoid this default behavior, script-local variables can be used as well. In Groovy, the keyword `def` must be used so `def sum = 0`. In that case, the `sum` variable is not stored as a process variable.

An alternative way to set process variables is done by explicitly using the execution variable that is available in the script task.

```
<script>
    def bookVar = "BPMN 2.0 with Activiti"
    execution.setVariable("bookName", bookVar);
</script>
```

As a final remark on some limitations while coding script, it is worth mentioning that some keywords cannot be used as variable names—`out`, `out:print`, `lang:import`, `context`, and `elcontext`—because these are reserved keywords within Activiti and Groovy. Back to our process—back to the `Check Credit` script task.

Implementing the credit check script task

The implementation of the `Check Credit` task is pretty straightforward. The Loan Sharks Company agrees to let a customer pass the credit check when his or her income is bigger than the requested loan amount. Check out listing 1 to see the BPMN 2.0 XML fragment that defines the script task.

Listing 1 Script task BPMN fragment

```
<scriptTask id="checkCredit" scriptFormat="groovy">                    #A
  <script>
    out:print "Checking credit for " + name + "\n";                    #1
    creditCheckOk = false;                                             #2
    if(income > loanAmount){
```

```
        creditCheckOk = true;
      }
      out:print "Credit checked for " + name + "\n";
    </script>
  </scriptTask>
```

#A The Groovy script language declaration
#1 Using the process variable 'name'
#2 Defining a new process variable

In the script, we use the name variable (#1) to print some logging on the console. Then, we create a new process variable (#2) that will hold the information about the credit check in a Boolean. As long as our loan requestor's income is bigger than the requested loan amount, the credit check will pass (#3).

Using script tasks on the Activiti Engine

To use the Groovy scripting engine on the standard Tomcat distribution that is installed with the Activiti installation, it's necessary to copy the Groovy groovy-all-version.jar to the `tomcat/lib` directory. You can find the Groovy jar in the examples/activiti-engine-examples/libs-runtime directory of your Activiti distribution.

We have our first script task in the process under control—moving on to the java service tasks.

Defining Java service tasks

Now, we are going to implement the `Check Risk` task and the `Create Application` task. The `Check Risk` task will return a Boolean indicating whether the risk of lending money to a certain customer is too high or not. The `Create Application` task gathers all the information produced so far in a `LoanApplication` Java bean and puts this bean on the process as a variable so we have easy access to it in the subsequent steps of the process.

Implementing the risk check Java service task

The `Check Risk` task is implemented with a Java service task. Typically, these kinds of checks contain valuable logic for the business and change frequently. To give more control to the business in maintaining this kind of logic and enable possible reuse for other applications, business rule engines are often used. Now, we'll use the Java service task to implement the check.

To illustrate how the risk check behaves, see listing 2.

Listing 2 Implementation of the Check Risk service task

```
public class RiskChecker implements JavaDelegation {                    #A

  public void execute(DelegateExecution execution) {
    String name = (String) execution.getVariable("name");
    System.out.println("Checking loan risk for : " + name);
    boolean riskCheckOk = false;
    if(!name.equalsIgnoreCase("Evil Harry")) {                           #1
      riskCheckOk = true;
    }
    execution.setVariable("riskCheckOk", riskCheckOk);                   #2
  }
}
```

#A Java service task standard interface
#1 Checking the name process variable
#2 Setting `riskCheckOk` variable on execution

In the `execute` method of the `RiskChecker` class, we check if the name of our loan applicant is `Evil Harry` (#1). We have to do this because the Loan Sharks Company knows Harry well enough to be sure that, whenever money is lent to him, nobody ever sees it back again. Harry will not pass the risk check! Don't forget to set the `riskCheckOk` variable on the execution (#2) so we can use it later.

Implementing the create application Java service task

The Create Application service task gathers all the data that was produced in the previous steps in one object instance and puts that instance in the process as a process variable. Code listing 3 displays the service task implementation.

Listing 3 The Create Application service task implementation

```
public class ApplicationCreator implements JavaDelegation {

  public void execute(DelegateExecution execution) {
    LoanApplication la = new LoanApplication();                    #1
    la.setCreditCheckOk((Boolean) execution
      .getVariable("creditCheckOk"));                              #2
    la.setRiskCheckOk((Boolean)
        execution.getVariable("riskCheckOk"));
    la.setCustomerName((String)
        execution.getVariable("name"));
    la.setRequestedAmount((Integer)
        execution.getVariable("loanAmount"));
    la.setEmailAddres((String)
        execution.getVariable("emailAddress"));
    execution.setVariable("loanApplication", la);                  #3
  }
}
```

#1 Creating the `LoanApplication` bean
#2 Retrieving process variable to populate the bean
#3 Setting the `LoanApplication` instance on the process

In the `execute` method of the `ApplicationCreator` Java service task class, we create the `LoanApplication` instance (#1). Remember that this object has to implement the `Serializable` interface; otherwise, the Activiti Engine will not be able to store its state in the process database. The values with which we populate the object (#2) are retrieved, on one hand, from the start form we will build and, on the other, the ones that have been set by the two checks we implemented in the earlier tasks. At the end, don't forget to store the variable in the execution (#3).

We saw that the `Check Credit` script task and the `Check Risk` service task can be executed in parallel so we take a look at that the parallel gateway BPMN construct.

Explaining the parallel gateway

The parallel gateway is used in the loan request process model to indicate that the check tasks can be executed independently and in parallel. The parallel gateway concept in BPMN is used to model concurrency in a process. It allows the execution path of a process to fork into multiple paths of execution or join multiple incoming paths together to a single point. The functionality of the parallel gateway is based on the incoming and outgoing sequence flow.

- `join`—All concurrent executions arriving at the parallel gateway wait in the gateway until an execution has arrived for each incoming sequence flow. Then the process continues past the joining gateway.
- `fork`—All outgoing sequence flows are followed in parallel, creating one concurrent execution for each sequence flow.

An important difference with, for example, the exclusive gateway is that the parallel gateway does not evaluate conditions.

Now that we have our business logic together and have talked about the control flow surrounding it, we will take a look at the resulting BPMN 2.0 XML.

Creating the BPMN 2.0 XML file

To be able to test this part of the loan request process, we are going to build a BPMN 2.0 XML file with all the tasks covered so far. We saw in the BPMN 2.0 model that the Check Credit task and the Check Risk task should be executed in parallel. The construct that is used in BPMN to realize this kind of behavior is the parallel gateway.

Take a look at code listing 4 to see how our loan request process looks like so far.

Listing 4 BPMN 2.0 XML for the partly finished loan request process

```
<process id="loanrequest" name="Process to handle a loan request">
  <startEvent id='theStart' />
  <sequenceFlow id='flow1' sourceRef='theStart'
      targetRef='fork' />
  <parallelGateway id="fork" />                                    #1
  <sequenceFlow id='flow2' sourceRef="fork"
      targetRef="checkCredit" />
  <sequenceFlow id='flow3' sourceRef="fork"                        #2
      targetRef="checkRisk" />
  <scriptTask id="checkCredit" scriptFormat="groovy">
    <script>
      out:print "Checking credit for " + name + "\n";
      creditCheckOk = false;
      if(income < loanAmount){
        creditCheckOk = true;
      }
      out:print "Credit checked for " + name + "\n";
    </script>
  </scriptTask>
  <sequenceFlow id='flow4' sourceRef="checkCredit"                 #3
      targetRef="join" />
  <serviceTask id="checkRisk"
      activiti:class="org.bpmnwithactiviti.chapter4.RiskChecker">
  </serviceTask>
  <sequenceFlow id='flow5' sourceRef="checkRisk"
      targetRef="join" />
  <parallelGateway id="join" />                                    #4
  <sequenceFlow id='flow6' sourceRef="join"
      targetRef="createApplication" />
  <serviceTask id="createApplication"
      activiti:class="org.bpmnwithactiviti.
          [CA]chapter4.ApplicationCreator">
  </serviceTask>
  <sequenceFlow id='flow7' sourceRef="createApplication"
      targetRef="wait" />
  <userTask id='wait' />                                           #5
  <sequenceFlow id='flow8' sourceRef="wait"
      targetRef="theEnd" />
  <endEvent id='theEnd' />
</process>
```

#1 Declaration of the parallel gateway fork
#2 One execution flow forking to a task
#3 After the task is finished join again
#4 Declaration of the parallel gateway join
#5 User task for testing purposes

After the start of the process, execution is brought to the parallel gateway (#1) by the first sequence flow. Execution forks and the `checkCredit` script task and `checkRisk` Java service task are executed concurrently (#2). After both these tasks have finished, the process is guided by the outgoing sequence flows of the tasks (#3) toward the join (#4). After that, the process continues normally with the `createApplication` task. The user task that is defined at the bottom of the xml (#5) is purely there for testing purposes. Without it, the process ends after the `createApplication` task and we cannot query it anymore to see the value of the process variables.

> **NOTE** In listing 4, we didn't use the `definitions` element. We just leave it out to be brief but remember that it is needed when we want to execute the BPMN 2.0 XML, whether we use it standalone in a unit test or on Activiti Engine after a deployment.

All the constructs used in the process so far can be easily tested in a unit test in Eclipse. It is good practice to test as early as possible. We want to get rid of possible bugs in the BPMN before we are deploying on Activiti Engine, so let's give our process a spin!

Testing the process with Eclipse

We will use the `ActivitiRule` class to get the `RuntimeService` and use the `@Deployment` annotation to deploy our process. Take a look at the code in code listing 5 to see how it is done.

Listing 5 Testing the loan request process tasks

```
public class LoanRequestTest {

  @Rule
  public ActivitiRule activitiRule =
      new ActivitiRule("activiti.cfg-mem.xml");                          #A

  @Test
  @Deployment(resources={"chapter4/loanrequest_firstpart.bpmn20.xml"})
  public void creditCheckTrue() {
    Map<String, Object> processVariables =                               #1
    ProcessInstance pi = activitiRule.getRuntimeService()
        .startProcessInstanceByKey(
            "loanrequest", processVariables);
    processVariables = activitiRule.getRuntimeService()
        .getVariables(pi.getId());
    LoanApplication la = (LoanApplication)                               #2
        processVariables.get("loanApplication");                         #2
    assertEquals(true, la.isCreditCheckOk());
    assertEquals(true, la.isRiskCheckOk());                              #3
  }
}
```

#A Configures Activiti to use the in-memory database
#1 Starts the process with a variables map
#2 Retrieves the `LoanApplication` process variable
#3 Tests the process variable

As you can see, we don't use the default `activiti.cfg.xml` but take a configuration file that uses the in-memory H2 database. We deploy the loan request BPMN 2.0 XML that we defined in listing 4 to do some early testing. Since we didn't implement a start form for the process yet, we have to start the process with some programmatically defined variables (#1). If we don't do this the service tasks will run into `NullPointerExceptions`.

Let's start a loan request for Miss Piggy. As you can see Miss Piggy earns more money than she wants to borrow so passing the checks shouldn't be any problem. You can see as well that she has an email address; this address can be used later to implement the email service task. After the process is started, we get the `loanApplication` variable out of the process (#2). This variable is set by the `CreateApplication` task. If the tests (#3) succeed, it means that all the tasks have run successfully.

Summary

You have seen how script tasks and Java service tasks can perform the logic that is needed to handle a loan request. We have seen how to implement a bit of business logic and test it with a simple unit test. We also covered two kinds of gateway that BPMN 2.0 provides, to control the paths of execution in a process, the exclusive gateway and the parallel gateway.